GTU - A workbench for the development of natural language grammars

Volk, Martin ; Jung, M ; Richarz, D

Abstract: In this report we present a Prolog tool for the development and testing of natural language grammars called GTU (German: Grammatik-Testumgebung; grammar test environment). GTU offers a window-oriented user interface that allows the development and testing of natural language grammars under three formalisms. In particular it contains a collection of German test sentences and two types of German lexicons. Both of the lexicons can be adapted to a given grammar via an integrated lexicon interface. GTU has been implemented in Prolog both under DOS and UNIX. It was originally developed as a tutoring tool to support university courses on syntax analysis but in its UNIX-version it allows for the development of large grammars.

GTU - A workbench for the development of natural language grammars[1]

MARTIN VOLK
University of Zurich
Institute of Computer Science
Winterthurerstr. 190
CH-8057 Zurich
+41-1-257-4325
volk@ifi.unizh.ch

MICHAEL JUNG, DIRK RICHARZ
University of Koblenz-Landau
Institute of Computational Linguistics
Rheinau 1
D-56075 Koblenz
+49-261-9119-469
richarz@informatik.uni-koblenz.de

Abstract

In this report we present a Prolog tool for the development and testing of natural language grammars called GTU (German: Grammatik-Testumgebung; grammar test environment). GTU offers a window-oriented user interface that allows the development and testing of natural language grammars under three formalisms. In particular it contains a collection of German test sentences and two types of German lexicons. Both of the lexicons can be adapted to a given grammar via an integrated lexicon interface. GTU has been implemented in Prolog both under DOS and UNIX. It was originally developed as a tutoring tool to support university courses on syntax analysis but in its UNIX-version it allows for the development of large grammars.

1 Introduction

Any computer system that analyses natural language input (be it a grammar checker or a tool for machine aided translation or the like) needs a formal grammar in order to map the input to a structure that groups it in some meaningful way. However, it is by no means clear how the grammar for such a system should look like nor how it is to be compiled. The tool presented in this paper has grown out of an effort to structure the grammar development process and to allow easy testing of such grammars. Although some other such tools have been developed over the years they were mostly designed for the use within a special NLP project.

Our primary goal therefore was the development of a flexible and user-friendly tool that would allow systematic development and testing of grammars in various formats. In fact, the first prototype was intended as a tutoring tool to supplement a syntax course in computational linguistics. We now have a full blown system interacting with a huge lexicon and a suite of test sentences which we call GTU (German: Grammatik-Testumgebung; grammar test environment). We support

[1]Many people have contributed to GTU. We would like to acknowledge the work of Arne Fitschen, Johannes Hubrich, Christian Lieske, Stefan Pieper, Hanno Ridder, Andreas Wagner, and Martin Wirt.

grammars under three formalisms: DCG (Definite Clause Grammar; cp. [PS87]) with feature structures, ID/LP (Immediate Dominance / Linear Precedence) grammars (cp. [GKPS85]), and LFG (Lexical Functional Grammar; cp. [KB82]). GTU also provides a first step towards semantic processing with LFG-style grammars.

Our prototype system has been developed on DOS-PCs with Arity-Prolog. It consists of 300 KByte of compiled Prolog code. We chose Prolog as the implementation language since we needed parsers of various types. On the one hand we wanted parsers to transform our linguistic style grammar rule notation into executable code. On the other hand we wanted parsers to process the natural language sentences with our grammars. We will hence call the first type grammar processors and the second NL-parsers to avoid confusion. In addition Prolog allows us to organize our test suite in a database format. Prolog has proven a successful candidate with the prototype and was therefore also chosen for the full implementation. We switched our hardware platform to UNIX workstations and we reimplemented GTU using SICStus Prolog. SICStus provided two important advantages. It allowed us to develop GTU under an X-window user interface and it provided external database facilities for the storage and flexible indexing of the test suite and the lexicon. GTU is now a stand-alone system consisting of 4.5 MByte of compiled Prolog code (not counting the lexicon). It has been tested in an inhouse-workshop and proven to be very robust. Figure 1 gives an overview of GTU's components. The modules in angled boxes are provided by the system. The modules in rounded boxes are generated during runtime. While the rule files (lexicon interface, syntax, and semantic rules) must be supplied by the grammar developer all the other modules in rounded boxes are automatically generated.

2 GTU's functionality

GTU runs on SUN-workstations SUN4 under X-Windows. X-Windows provides a powerful graphical user interface for UNIX machines. With this setup GTU allows the grammar developer to perform the following operations in dedicated windows:

- to work on multiple grammar files,
- to add sentences to multiple test suite files,
- to check the sentence structure as given by the NL-parser,
- to manually feed the NL-parser with sentences (or parts thereof) that are to be analysed,
- to select sentences from the test suite by simply clicking on them and feed them to the NL-parser,

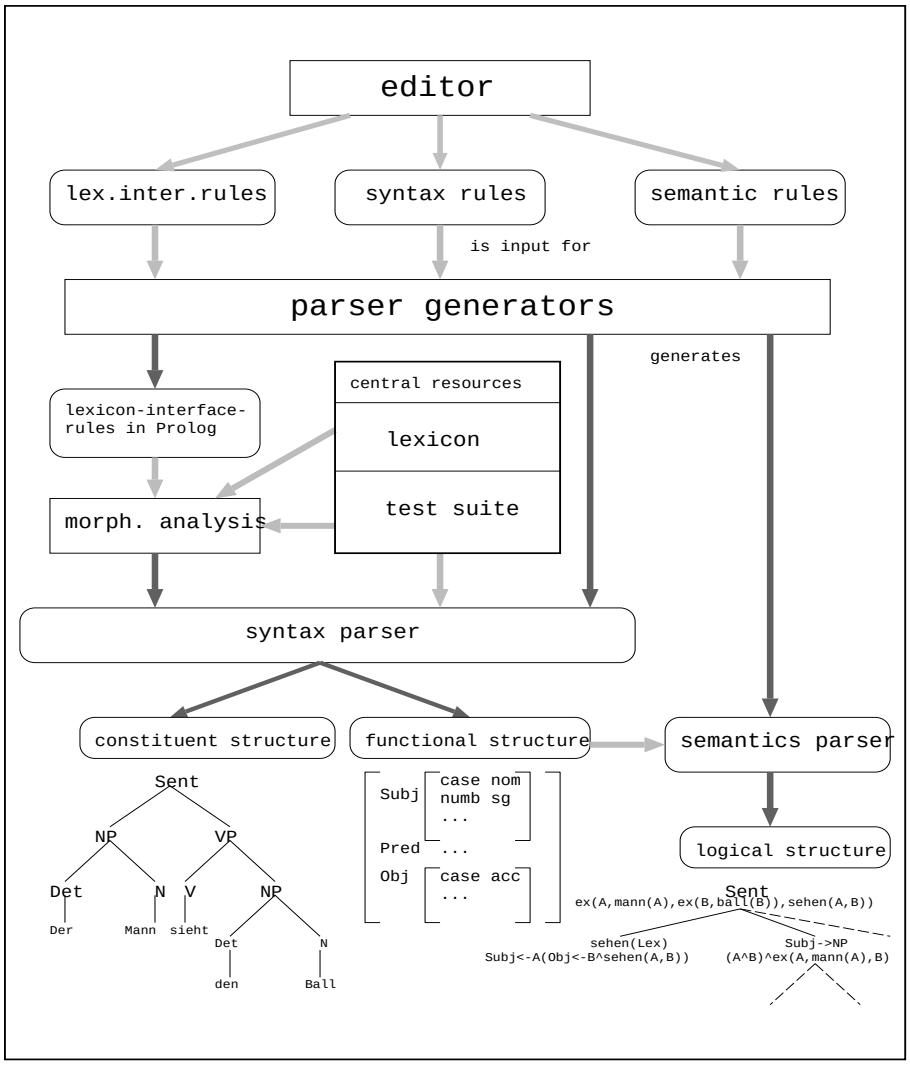

Figure 1: Overview of GTU's system structure

- to provide lexical information on given word forms,
- to switch between different formalisms, tracing and output modes.

The following paragraphs briefly describe some aspects of GTU. Lexicon and test suite will be subsequently described in special sections.

Modularising the grammar

When developing a large grammar it is inevitable to separate it into modules. GTU supports grammar modularisation into files that can be separately loaded and tested. For example, we organize the NP-syntax and the VP-Syntax in different

files. These can be separately tested by entering NP constituents (*der Mann* engl.: the man) or VP constituentes (*sieht den Ball* engl.: sees the ball) into a special window for manual test input. It is also possible to delete a grammar module or a part thereof from the loaded grammar.

Translating the grammar

The various grammar processors (there is one for each formalism) translate grammar rules given as ANSI texts into Prolog code. This code, in connection with the lexical rules generated by the morpho-lexical module, constitutes the parsing code for the analysis of the natural language input.

The grammar processors are themselves parsers that scan the grammar files, tokenize the grammar rules and translate them into the desired format, so that they can serve for parsing the test sentences. For efficiency reasons the processors are designed as SLR parsers. Because of the complex syntax of the grammar rules (cp. section 5) the parsing table for one processor consists of more than 150 states.

For the different formalisms, different types of NL-parsers are produced. An ID/LP grammar is treated with a bottom-up chart parser whereas the parsers for DCG and LFG grammars are top-down depth-first parsers (i.e., parsers that work in lockstep with the search strategy of Prolog's theorem prover).

GTU's NL-parsers keep track of the constituents being recognized. Thus, in case of failure to parse the complete sentence, partial results can be retrieved. This helps in locating grammar errors.

Static grammar checks

Because some structural properties of a grammar can corrupt the proper functioning of particular NL-parsers (by driving them into infinite loops), GTU provides for static grammar checks. In an ID/LP grammar LP-rules are transitive by definition. It is therefore prohibitive to have cycles within the LP-rules. GTU therefore offers a static check for cycles within LP-rules. If there are such cycles they are presented to the grammar developer for correction. In a similar manner we can detect left recursive rules for the grammars being processed by depth-first parsers as well as cycles within alias definitions.

Help in grammar development

Grammar development is a complex and error prone process. And although GTU claims to be a user-friendly and easy-to-learn system there will always be situations where the grammar developer needs a particular information. GTU includes a hypertext help module. All help files are marked up with HTML, the Hypertext

Markup Language of the World Wide Web. They can thus be searched with any WWW browser. GTU help provides information on the following topics:

- the structure and information of the lexicon
- the syntax for the lexicon-interface rules
- the syntax of the grammar rules for the different grammar formalisms
- the structure and usage of the test suites
- the handling of the editors
- the handling of the GTU desktop

Integrated editors

Although it is possible to use any editor desired within a multi-windowed system there are advantages in having special purpose integrated editors in a system like GTU. We envision the grammar development process as an incremental process where test steps and modification steps follow each other rapidly. Therefore we provide a grammar editor that allows for saving, translating, and loading the grammar under one mouse click. The grammar is immediately ready for testing. There is another editor for modifying the test suite files with analogous functionality. Both editors support the usual search and replace functions as well as copy and paste operations.

Visualising parsing results

One of GTU's main features is the user-friendly presentation of parsing results. For all of the formalisms, the parse tree can be displayed tree-like (see figure 2). In addition there are formalism dependent output facilities. For LFG grammars GTU outputs the functional structure (F-structure) of a sentence, thus giving grammatical functions such as subject, predicate and object. For DCGs and ID/LP grammars the parse tree is presented as an indented structure with all the features defined in the grammar (see lower half of figure 3).

Output can be directed by the grammar developer into one window, into multiple windows, or into a file. The multiple window option allows to contrast the computed structures on the screen. For convenience, parsing results (regardless of the format chosen for display) can be stored in files. This makes it easy to use them for documentation and evaluation purposes. When printing the parsing output to a file we provide two options. First there is a standard ANSI format, which approximates a tree structure with standard ANSI characters. Second there is a

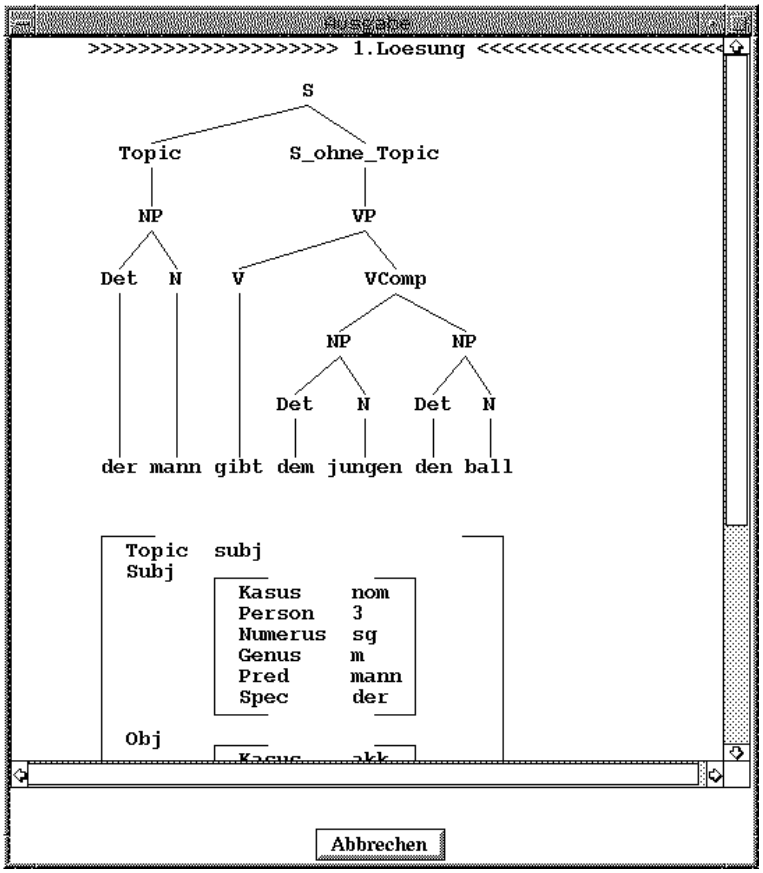

Figure 2: Parser Output under the LFG-formalism

LaTeX format which leads to nicely printed trees using the *QTree Macro Package*[2]. This macro gets the tree as a nested structure in postfix format. An example of a tree printed in this fashion is depicted in figure 4.

Tracing the parsing process

The parsing of natural language input can be traced on various levels. It can be traced during the lexicon lookup process thus providing the morphological information for every word form as stored in the lexicon. It can also be traced during the evaluation of the lexicon interface rules. With this option the grammar developer can observe which lexical rules are generated for a given word form. Furthermore, parsing can be traced during the application of the grammar rules. For the ID/LP formalism GTU presents every chart edge produced by the bottom-up chart parser. For the DCG and LFG formalisms GTU shows CALL, FAIL, REDO, and EXIT

[2]The *QTree Macro Package* has been developed by J.M. Siskind and A. Dimitriadis and can be obtained via the email list Ling-Tex@shsu.edu.

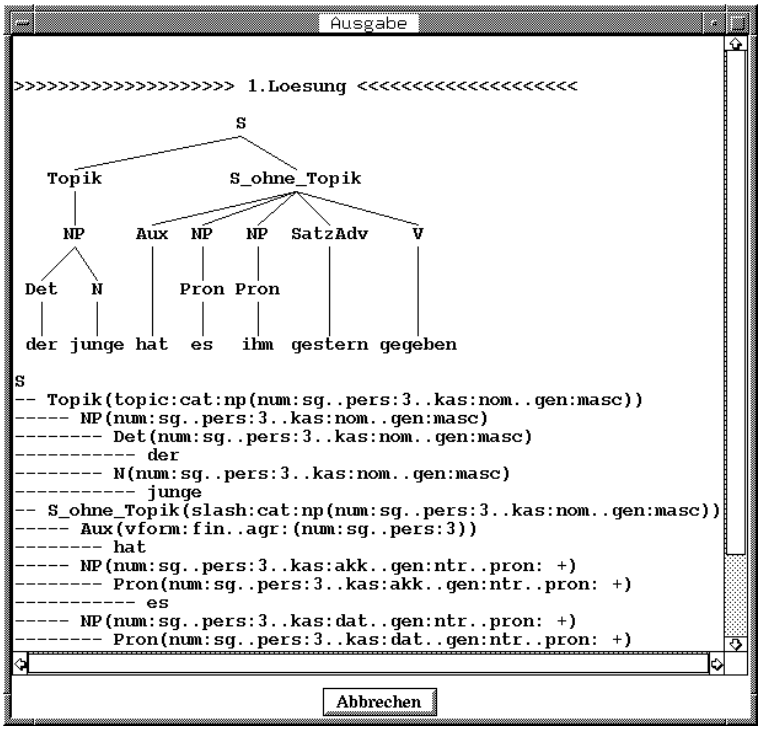

Figure 3: Parser Output under the ID/LP-formalism

ports for a predicate. If semantic processing is enabled these ports are also shown for the semantic rule predicates.

3 Test suite

GTU contains a test suite of approximately 350 German test sentences organized in two different ways. A test suite is defined as a selected collection of syntactically well-formed natural language sentences and contrastively selected ungrammatical sentences (termed: non-sentences). The non-sentences can be used to check for overgeneration of the grammar. The collection is being built according to the following principles:

1. The vocabulary for the sentences and non-sentences is derived from a controlled set of words.

2. The sentences and non-sentences are annotated with respect to their syntactic properties. The annotation tries to be as theory-neutral as possible.

3. The sentences differ among each other in at least one feature. The same holds for the non-sentences.

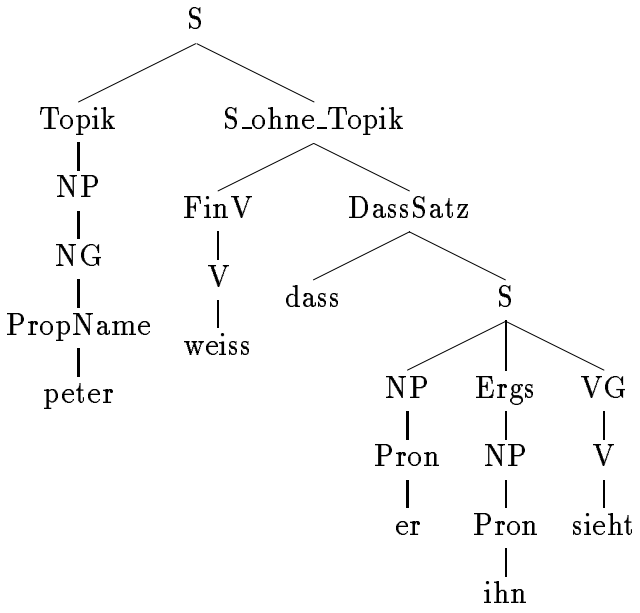

Figure 4: Printed form of a tree produced by GTU for the sentence *Peter weiss, dass er ihn sieht.* engl.: Peter knows that he sees him.

4. Sentences and Non-sentences can be subclassified into classes of test sentences. A class is defined by some feature that holds for all its members.

5. A non-sentence is identical to a sentence up to one feature. This particular feature is the reason for the ungrammaticality.

Our **first test suite** setup organizes the sentences into 15 classes, each pertaining to a specific collection of syntactic phenomena. The classes contain about 70% sentences and 30% non-sentences. There are, for example, classes for sentences displaying differently complex noun phrases, sentences containing relative clauses, and sentences involving negation. Prolog terms are used to connect the sentences to information about their grammaticality and their syntactic properties. In the following some examples are given (the first argument of every fact pertains to the grammaticality of the sentence, the second to the words of the sentence, and the third is a list of pointers to syntactic properties: **npKon** for congruency within the NP, **svKon** for congruency between subject and verb, **it** for intransitive verb and so on). The last two of these examples are ungrammatical due to violations of subject-predicate congruency on the one hand and valency on the other.

```
t(grammatisch,"Der Mann schlaeft.",[npKon,svKon,it]).
t(grammatisch,"Der Mann sieht den Ball.",[npKon,svKon,tr]).
t(grammatisch,"Der Mann hilft dem Jungen.",[npKon,svKon,dat0]).
t(ungrammatisch,"Der Mann schlafen.",[npKon,svKon,it]).
t(ungrammatisch,"Der Mann schlaeft den Ball.",[npKon,svKon,it]).
```

At the heart of the **second test suite** setup is a tree-like graph structure, depicted in figure 5, whose 27 inner nodes represent syntactic phenomena like coordination and valency. We therefore call this structure a phentree (phenomena tree). Inner nodes down in the phentree inherit all features from the phenomena on the path that connect them to the root of the tree. Thus, an inner node near the top is more general than one near the bottom. This inheritance provides a means to represent information more compact, and makes it easier to keep it consistent.

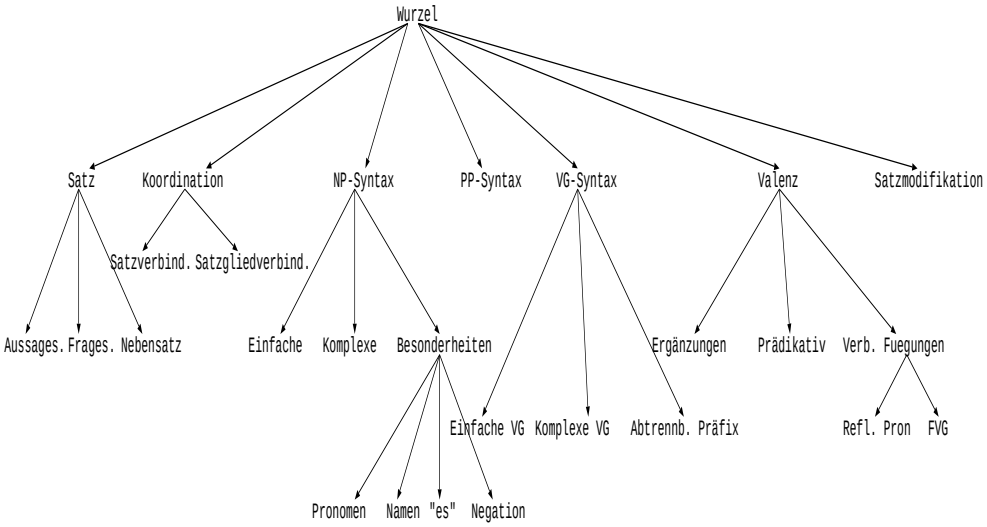

Figure 5: The test suite's syntactic phenomena tree

Sentences are attached to the phentree's leaves. It is legitimate to attach a sentence to more than one leaf. That means that a sentence can be a representative for different phenomena. The sentences are stored in the SICStus-Prolog external database while a special user interface allows to traverse the phentree in all directions. For every node in the tree the interface provides for a display of all sentences it subsumes. It is also possible to select multiple nodes at the same time to obtain the intersection of the sentence sets subsumed by the selected nodes. On the other hand it is possible to select the sentences by feature.

In order to organize the sentences into the phentree we needed a text description language. We chose SGML (Standard Generalized Markup Language) since it is a flexible database language for textual documents. We developed a so called document type definition (DTD) as grammar for the SGML markup. Then all sentences were marked with SGML and validated with an SGML parser. The parser output (a structured SGML document containing all our test sentences) was translated into our Prolog database. In this way we have a transparent markup scheme, which allows easy interchange of our sentences with other projects and we get free consistency checks performed by the SGML parser (for more details see [Vol94] or [FP94]).

4 Morphological analysis and lexicon

Every syntax analysis of natural language sentences is critically dependent on lexical information. The lexicon is the repository of morpho-syntactic information for every word form and the grammar only serves to assemble this information in a systematic and fixed way.

4.1 GTU's handcoded stem lexicon

Since its prototypes GTU has used a handcoded German lexicon with about 240 word stems, the vocabulary of which is carefully tailored towards the test suite. This lexicon contains stem information and affix information for the open word classes (nouns, verbs, adjectives). A given word form is divided into stem and inflectional affixes. The information found under the stem and those found under the affix are unified and presented to the lexicon interface. Word forms from closed word classes (determiners, adverbs, prepositions, pronouns etc.) can be looked up directly. All available readings of a word form are produced before parsing starts.

4.2 GTU's interface to a lexical database

Recently the CELEX [BPvR93] database was interfaced to GTU. CELEX is a database commercially available on CD-ROM and offers lexical information for English, Dutch and German. Only the German part, restricted to morpho-syntactic information, was used for GTU (ignoring e.g. phonological features). In order to speed up the lexical lookup we discarded the division of morphological analysis and stem lookup in favor of a full form database thus reducing lookup to pure search and inheritance. In order to keep the amount of information manageable we divide the word form information into specific and general information. A given word form gets all its specific information and additionally inherits all the general information from its lemma. Given the verb form *sieht* (engl.: sees) the information about person and number is stored as word form specific information together with a pointer to the lemma. Following this pointer we find general information for the word like the type of perfect auxiliary or valency.

The lexical information for the 360.000 word forms is compiled into a SICStus Prolog external database, separated in modules according to the first letter of every word form. The 26 parts of the database need a total of 45 MBytes of memory but thanks to the efficient indexing scheme the lookup times are very fast. On a SUN4 we receive all the lexicon information for a given word form within a second. [Lie94] contains a detailed description of the GTU-to-CELEX interfacing.

4.3 Flexible lexicon interface for different grammar formalisms

A core concept of our development environment is the flexible lexicon interface that mediates between every grammar and the lexicon. The interface rules are needed since every grammar formalism needs the lexical information in a different format. This is most apparent with valency information. While LFG needs a list of complements (e.g. for the verb *finden* a list with a nominative subject and an accusative object), other grammars need a numerical pointer to a valency class (e.g. `finden(valency=10)`). With the lexicon interface rules the grammar developer can specify what information the grammar needs out of the lexicon, in which format the information should be structured and how it should be named.

While the lexicon structure is hidden from the grammar developer, she receives the following information to help in setting up the lexicon interface rules:

- parts of speech used in the lexicon,
- features and their domains for every part of speech,
- the classification of words into parts of speech,
- the morphological information for a given word form.

Let us demonstrate with an example how the lexicon interface works. Let's say we want to interface determiners to a given ID/LP grammar. From system help (as well as from the handbook) we learn that GTU provides the following information for German determiners.

feature	feature name	domain
part of speech	wortart	det
number	numerus	{sg,pl}
case	kasus	{nom,gen,dat,akk}
gender	genus	{m,f,n}
adjective inflexion	flex	{stark,schw,gem}

Let's say that we would like to use determiners under the name `Artikel` with the information given in the lexicon under case, number and gender. The information for adjective inflexion be of no interest for the sake of this example. Then we can use the following interface rule.

```
if_in_lex (wortart = det) then_in_gram
    Artikel[kas = #kasus, num = #numerus, gen = #genus].
```

This states that a word form recognized as a determiner can be used as `Artikel` in our current grammar. Furthermore it states that the information obtained for

case **#kasus** will be assigned to the feature named **kas** in our current grammar (and the like for number and gender). For the determiner *der* (engl.: the) the lexicon interface routine would thus create the following lexical rules for the grammar. These can be understood as rules in a DCG-like format except that the arguments in brackets are feature structures that are internally represented as open ended lists (like in GULP; cp. [Cov89]).

```
Artikel[kas = nom, num = sg, gen = mask] -> der.
Artikel[kas = gen, num = sg, gen = fem]  -> der.
Artikel[kas = dat, num = sg, gen = fem]  -> der.
Artikel[kas = gen, num = pl]             -> der.
```

Just to show how differently the lexicon information can be organized we present an analogous interface rule for the LFG-formalism where feature passing is notated with an up-arrow (^).

```
if_in_lex (wortart = det) then_in_gram
          Artikel, (^ Kas)  = #kasus,
                   (^ Num)  = #numerus,
                   (^ Gen)  = #genus.
```

In general the expression following **if_in_lex** defines the test criterion to be checked against the word's lexical information. The expression after **then_in_gram** defines how the information shall be mapped to a syntactic category in case the test criterion succeeds.

5 A typical grammar development session

A working session begins with the GTU desktop (see figure 6). All system functions and options can be reached from there.

The first step in grammar development is the writing of grammar rules for a specific syntactic phenomenon. In GTU this involves choosing a set of test sentences from the test suite, opening a grammar editor and writing the grammar rules. The grammar rule notation aims at being concise and as close as possible to linguistic conventions. Typical DCG rules are:

```
(1) S          -> NP[X]  VP[X]  |  X = [kas=nom].
(2) NP[kas=K]  -> Det[kas=K, num=N, gen=G]  N[kas=K, num=N, gen=G].
```

Rule (1) says, that a constituent of type **S** consists of a constituent of type **NP** followed by **VP**. The feature structures are given in square brackets. A capital

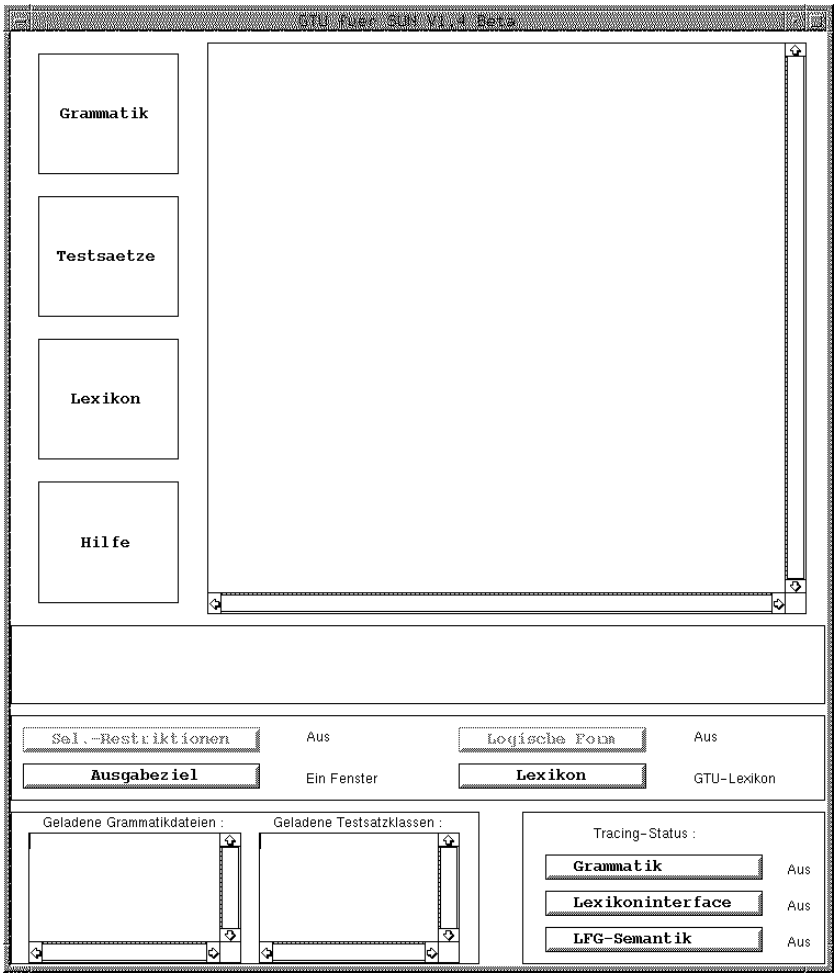

Figure 6: GTU's desktop

letter in a feature structure represents a variable. Identical variables within a rule stand for identical values. Hence the feature structures for `NP` and `VP` in rule (1) are declared to be identical. In addition the feature structure equation behind the vertical bar `|` specifies that this `X` must be unified with the feature structure `[kas=nom]`. Rule (2) says that an `NP` consists of a `Det` followed by an `N` and that the features `kas`, `num` and `gen` are set to be identical while only the feature `kas` is passed on to the `NP`-node.

There are further provisions for optional constituents, terminal symbols within a grammar and a reserved word representing an empty constituent.

In an ID/LP grammar one needs to specify ID-rules and LP-rules. While the ID-rules look much the same as DCG rules with the exception that there are commas between the constituents on the right hand side, as is usual in the linguistics literature. The LP-rules state precedence relations with the `<` symbol. Here is one example for each:

```
(3) S          -> NP[X],  VP[X]   |  X = [kas=nom].
(4) NP < VP.
```

For the LFG-formalism we also tried to follow the linguistic usage as close as possible which leads to a less transparent rule format, because the usual symbols for the metavariables (up-arrow and down-arrow) are not available in the standard ANSI character set. Some examples:

```
(5) S          -> NP[(^Subj)=v, (v Kasus) =c nom]  VP[^=v].
(6) NP         -> Det[^=v]  N[^=v]
```

The up-arrow was substituted with the `^` symbol and the down-arrow with a lower case `v`. Accordingly rule (5) says that an `S` consists of an `NP` and a `VP`. The `NP`'s feature structure adds the information that the `Subj` feature of the mother node (i.e. of the `S`) is identical with the `NP`'s feature structure and that the feature `Kasus` of the `NP` must be set to `nom`. The `=c` operator is called constraint equation and stands for a non-unifying equation check.

Within GTU the grammar formalism is chosen by giving a certain extension to the name of the grammar file (e.g. an LFG grammar file must have the extension `.lfg`). When a grammar file is loaded the processor first checks for the formalism and then examines the rules accordingly. Rules with erroneous syntax are being reported to the grammar developer and ignored for the transformation, but all syntactically correct rules are transformed and stored in the Prolog knowledge base so that they can be tested.

When the grammar rules are arranged, the grammar must be connected to the lexicon via lexicon interface rules in a second step (cp. section 4.3). Although it is possible to have the grammar rules and the lexicon interface rules in the same file, it is highly recommended to treat them as different grammar modules in different files.

In a third step the grammar can be tested against test cases. In the beginning of the test phase we typically enter constituents manually to ensure the correct parsing of the sentence parts. Then we open a menu that displays all currently loaded test sentences and we select some for testing. If any of these tests leads to unexpected results the grammar rules must be modified. If eventually all selected test cases are being parsed satisfactory we initiate a complete test of all loaded test sentences. If these are also treated correctly we rerun this test and we redirect the output to a file for documentation purposes. In subsequent steps the grammar can be extended to more phenomena or a grammar under a different formalism can be developed for comparison with the first grammar.

6 Doing natural language semantics with GTU

According to its original design GTU was to be a pure syntax tool. Recently we have added facilities for computing the semantics of a sentence in the form of a logical expression. Thus for a sentence like *Der Mann sieht den Ball* GTU may compute a logical expression like `exist(A,mann(A),exist(C,ball(C),sehen(A,C)))`. Since the semantic interpretation is dependent on the syntactic structure, it is not possible to equip GTU with a routine that automatically produces a logical expression for any sentence. Sentence structures under GTU vary according to the given grammar. Instead we provide a language where the grammar developer can specify semantic rules for the grammar. The language currently works on F-structures of LFG grammars.

Let us demonstrate how this works. We assume that we have computed the following F-structure for the sentence *Der Mann sieht den Ball*[3]:

Topic	*subj*	
Subj	Kasus	*nom*
	Person	*3*
	Numerus	*sg*
	Genus	*m*
	Pred	*mann*
	Spec	*der*
Obj	Kasus	*akk*
	Person	*3*
	Numerus	*sg*
	Genus	*m*
	Pred	*ball*
	Spec	*der*
Pred	*sehen([Subj,Obj])*	

We may then specify the following semantic rules meaning that the semantics of the sentence (`Satz` as a keyword) is derived from the semantics of the predicate (`Pred` as a toplevel feature of the F-structure).

```
Satz <= X :
        Pred <= X.
```

The semantics of an NP as given in the following example is built up by first evaluating the features `Pred`, `Spec`, and optionally `Mod` (as indicated by the parentheses). The results are assigned to the lambda expressions `X^PredCont`, `X^AdjCont`,

[3] F-structure printout is also organized via LaTeX using a macro named *covingtn.sty* written by Michael Covington, University of Georgia.

and the variable `Y` respectively. The final logical expression for the NP is described in the top line between `<=` and `:` assigning the lambda expression `X^(PredCont && AdjCont)` to `Y`.

```
NP   <= Y <- ( X^(PredCont && AdjCont)) :
        Pred <= X^PredCont
        Spec <= Y
        (Mod) <= X^AdjCont.
```

This rule can only be applied to F-structures containing at least the features mentioned in the rule. Otherwise the evaluation fails (assuming no other `NP` semantic rule is defined). In addition to the evaluation of internal features it is possible to define constraints testing the existence of features. Also sets of feature-value pairs can be build.

Since several grammatical functions have the same internal representation because of identical syntactic origin (`Subj` and `Obj` are both derived from a noun phrase), we provide for the definition of aliases in order to reduce the number of semantic rules.

```
alias Subj NP.
alias Obj  NP.
```

These two alias rules denote that the evaluation of both `Subj` and `Obj` will be defined by a new constituent `NP` for which semantic rules are defined in this case.

For the above example sentence the semantic parser yields the output in figure 7. The structure of the tree describes the way the semantic rules and their internal definitions were used and what kind of logical expressions were created locally. `->` stands for the use of an alias, the keyword `Lexikon` shows that the logical expression has been derived from a semantic lexicon interface rule for the specified word.

Such semantic lexicon interface rules are similar to the syntactic interface rules except that the logical expression is defined instead of a category name (cp. chapter 4.3). The logical expression may contain lexical variables using the operator `#`.

```
if_in_lex (wortart = substantiv) then_in_gram X^#Lemma(X).
```

This interface rule defines a semantic lambda expression (indicated by the `^` operator) depending on a variable `X` for every noun (`Substantiv`). The variable `#Lemma` contains the lemma form of a noun.

Syntax parsing is independent of the semantic processing. It can be performed on LFG grammars ignoring any semantic rules. Development of the syntax and semantic rules can thus be organized in subsequent steps which is an important requirement for the modularisation of the grammar engineering process.

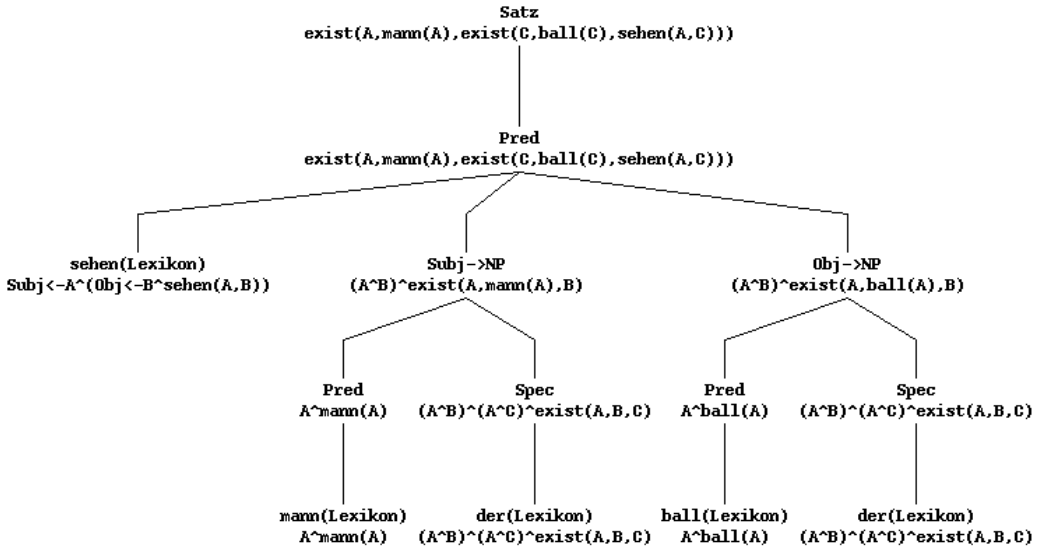

Figure 7: Derivation tree for a logical expression of the sentence *Der Mann sieht den Ball.*

7 Comparison with other systems

There are numerous publications describing grammar development tools. In table 1 we briefly present some well known examples. [Bac94] gives a much more detailed survey for some of these systems.

These systems differ widely in usability, scope and intention. Some systems serve as development environment for grammars under one specific theory (Alvey-GDE, ProGram or TAGDevEnv). Others allow the development of arbitrary grammars within the unification paradigm (ALE, ELU, STUF-WB). GTU supports the development of grammars under multiple grammar formalisms but is not set up for general unification grammars.

While some systems are designed for grammar development within one project, others are built for the experimentation with grammars. The Alvey-project aimed at compiling a large grammar for English and therefore the Alvey-workbench is particularly suited for this task. On the other end of the spectrum, Pleuk or GTU are aiming at experimentation and comparison with grammars. It is striking that the most flexible of these systems that rely heavily on unification are written in Prolog.

The workbenches can also be classified according to their overall capability as well as ergonomical and empirical criteria. Capability differs in number and types of parsers, types of consistency checks, and test organisation available in the system. Some of these criteria are mentioned in the right-most column of table 1. Some systems contain parsers that can be parametrized (e.g. Alvey-GDE and

Name	Place	Source	Programming Language	Grammar Formalism	System Features
ALE	Carnegie Mellon	[Car92]	Prolog	HPSG, DCG, Unification Grammars	typed feature structures
Alvey-GDE	University of Cambridge	[CBG91] [BCB88]	LISP	GPSG	parametrisable Parsers
ELU	ISSCO Geneva	[Est90]	LISP	Unification Grammars	Transfer module for Machine Translation
GTU	University of Koblenz	[JRV94]	Prolog	LFG, DCG, ID/LP	Integrated Test Suite
GWD	University of Nijmegen	[NKDvZ92]	n.k.	Affix Grammar over a Finite Lattice	incremental Consistency checks;
Language Craft	Carnegie Group	[Lan87]	LISP	Caseframes	part of a commercial Product for building NL-Interfaces; different rule types
META	Universita di Roma II	[MPPV91]	Prolog	Augmented Contextfree Grammar	consistency checks; interface to semantics
Metal-WB	University of Texas	[Whi87]	LISP	Augmented Phrase Structure Grammar	Transfer module for Machine Translation; Tool for lexicon development
Pleuk	University of Edinburgh	[CH93]	Prolog	Unification Grammars	Generator, user specifiable formalisms
ProGram	University of Sussex	[Eva85]	Prolog	GPSG	quasi nat. lang. command interpreter
STUF-WB	LILOG IBM-Stuttgart	[DR91]	n.k.	Unification Grammars	typed feature structures
TAGDevEnv	University of Saarbrücken	[Sch88]	LISP	Tree Adjoining Grammar	graphics based tree editors; consistency checks
TDL/UDiNe	University of Saarbrücken	[Bac94]	LISP	HPSG	typed feature structures
TFS	University of Stuttgart	[Bac94]	LISP	HPSG, LFG, GB	typed feature structures

Table 1: Survey of Grammar Workbenches

STUF-WB). Consistency checks are very differently elaborated. They are a main part of the GWD system where left-recursion, cyclicity, LL(1)-property and non-consistent calls can be detected. Grammar testing setup is mostly underdeveloped in these systems. Some (like Alvey) allow testing the grammar against corpora, but the integrated test suite is the very feature of GTU.

Ergonomical criteria depend mostly on the design of the user-interface. In particular we must judge the input facilities (editors and command languages) and the output facilities (views on the grammar, visualisation of the parsing results). Recent systems use graphical interfaces towards these goals (GTU, Pleuk, STUF-WB).

Empirical criteria relate to measurable factors in grammar development. This includes average parsing time for a sentence in relation to grammar size, or as [MPPV91] suggest, grammar development time in person-months. To our knowledge there are currently no empirical comparisons available.

8 Future Prospects

We have a vision of gradually extending GTU into a tool that allows the development of full-fledged natural language processing systems incorporating state-of-the-art formalisms and technology. At the same time we would like to keep GTU a robust, user-friendly, well-structured, and well-documented tool that can be used to support university level teaching in Computational Linguistics. We are well aware that it will become increasingly difficult to entertain both of the goals. We are measuring our progress towards the first goal against systems like the Core Language Engine [Als92] which provides a multi-level system with a broad coverage translating English sentences into a semantic representation. We check the second goal annually in courses at the University of Koblenz by having students fill in questionaires about the pros and cons and their satisfaction with GTU.

First steps towards extending GTU's capabilities are already under way:

- We need a facility to systematically evaluate parsing results. For a test run of several dozen test sentences it is very cumbersome to check the generated sentence structures manually. We are in the process of developing a tool that provides statistics on parsing results in tabular form. This includes: How many structures are generated for a given sentence? Are there any structures given for non-sentences? This tool will also enable the user to save an acceptable sentence structure to file and automatically compare the results of subsequent test runs against these files. We want to do that on three levels of granularity: First on the purely structural level (ignoring constituent and feature structures), second on the structural and constituent level and third including the feature structure level.

- We aim at increasing our sentence semantic processing by incorporation of a semantic ontology. We have set up a hierarchy of semantic features in the manner shown in figure 8.

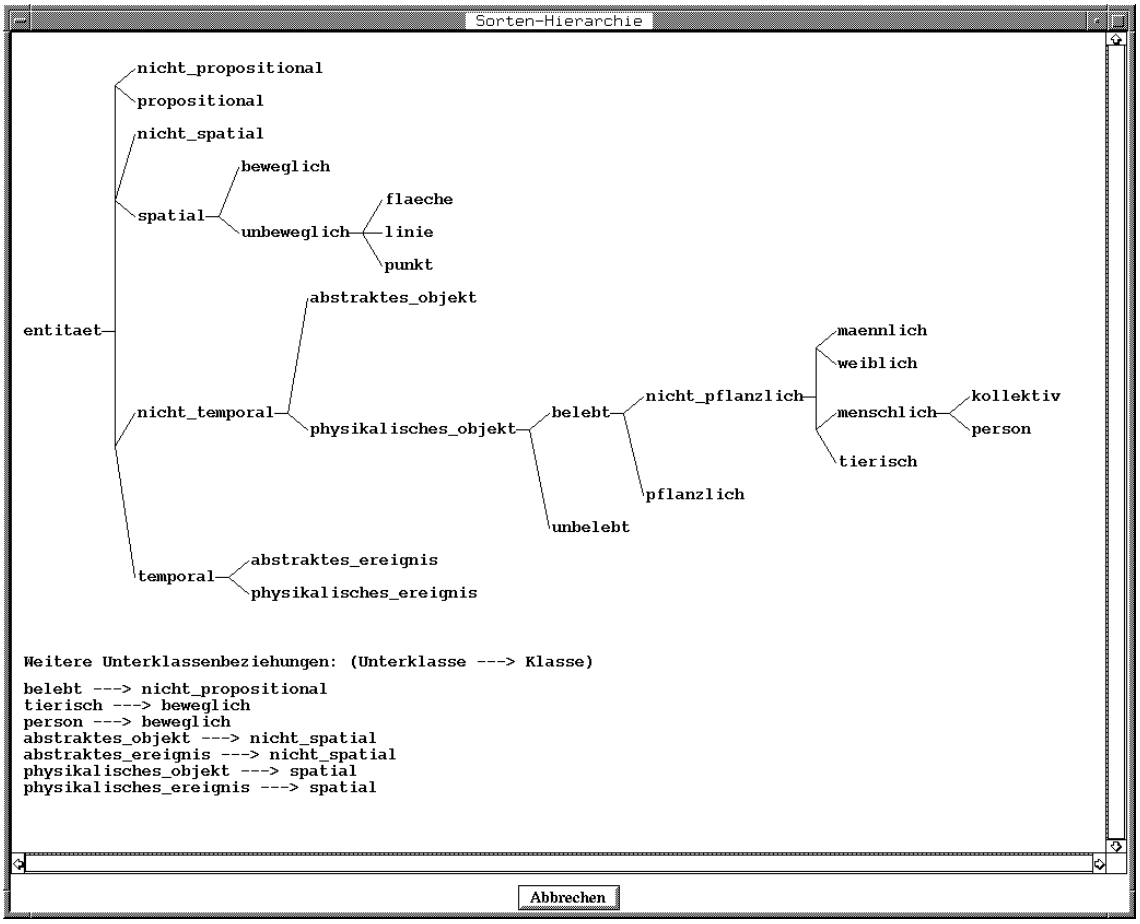

Figure 8: Hierarchy of semantic features

We have annotated the verbs in the GTU lexicon with selectional restrictions from this hierarchy. Nouns and Adjectives have been marked with semantic features describing their properties. This information is needed in order to filter out semantic anomalies from syntactically well-formed sentences. With these restrictions we determine that a sentence like "The flower finds the truth" is regarded as semantically ill-formed. More precisely, we compute the degree of anomaly by checking the distance of the selectional restrictions within the semantic hierarchy.

As future prospects for GTU we think of the following extensions:

- Support for more grammar formalisms: in particular HPSG (Head-driven

Phrase Structure Grammar) [PS94] and the inclusion of typed feature structures.

- Improvements in the user-interface: e.g. selective output (zooming in) of analysis results.
- Support for more grammar engineering principles: Data encapsulation within grammar modules; help in locating grammar problems during development (see [Hub93] for some preliminary results in a GTU prototype); prescriptions for the documentation of grammars.

Prolog has proven to be the language that served our needs for parsing, user interface design, and database applications. Various parsers have been integrated into GTU ranging from general SLR parsers to special purpose ID/LP chart parsers. The user interface was developed using almost any feature provided by the SICStus Prolog X-Windows predicates. We use text editors, graphic browsers, dialog boxes, pop-up menus, scrolling windows and status buttons. And although some irritations still remain (e.g. the cursor keys don't work in the editor) most of these widgets work perfectly. For our database purposes we rely on the SICStus Prolog external database facilities which turned out to be a very efficient way of accessing even large amounts of data. We believe Prolog to be unique in offering this range of functionality in an integrated way. As an addon we have developed a tool to automatically extract the call relations between predicates (p_0 calls $p_1 ... p_n$ as well as q_0 is called by $q_1 ... q_m$) to supplement our written documentation. This tool was developed and debugged within two days time which speaks for Prolog's fast prototyping in general and its meta-programming facilities in particular.

We would like to share GTU with other research groups and universities. The software accompanied by a manual can be obtained from the University of Koblenz for a nominal fee.

References

[Als92] Hiyan Alshawi, editor. *The Core Language Engine.* ACL-MIT Press Series in Natural Language Processing. MIT Press, Cambridge,MA, 1992.

[Bac94] Report of the EAGLES workshop on implemented grammar formalisms (March 1993). Technical report, DFKI, Saarbrücken, 1994.

[BCB88] B. Boguraev, J. Carroll, and T. Briscoe. Software support for practical grammar development. In *Proceedings of COLING*, pages 54–58, Budapest, 1988.

[BPvR93] R. H. Baayen, R. Piepenbrock, and H. van Rijn. The CELEX lexical database (CD-ROM). Linguistic Data Consortium, University of Pennsylvania, 1993.

[Car92] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, Cambridge, 1992.

[CBG91] John Carroll, Ted Briscoe, and Claire Grover. A development environment for large natural language grammars. Technical report, University of Cambridge Computer Laboratory, 1991.

[CH93] J. Calder and K. Humphreys. Pleuk overview. Technical report, University of Edinburgh. Centre for Cognitive Science, 1993.

[Cov89] Michael Covington. GULP 2.0: an extension of prolog for unification-based grammar. Research Report AI-1989-01, AI-Programs, The University of Georgia, Athens,GA, 1989.

[DR91] Jochen Dörre and Ingo Raasch. The STUF workbench. In Ottheim Herzog and Claus Rainer Rollinger, editors, *Text Understanding in LILOG*, pages 55–62. Springer, Berlin, 1991.

[Est90] Dominique Estival. ELU user manual. Technical report, ISSCO, Genf, 1990.

[Eva85] Roger Evans. ProGram - a development tool for GPSG grammars. *Linguistics*, 23(2):213–243, 1985.

[FP94] Arne Fitschen and Stefan Pieper. WAITS - Weiterentwickelte Alternative Testsatzsammlung. (Studienarbeit) Universität Koblenz-Landau, 1994.

[GKPS85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized phrase structure grammar.* Harvard University Press, Cambridge,MA, 1985.

[Hub93] Johannes Hubrich. Unterstützung der Fehleranalyse bei der Grammatikentwicklung innerhalb von GTU. (Studienarbeit) Universität Koblenz-Landau, 1993.

[JRV94] Michael Jung, Dirk Richarz, and Martin Volk. GTU - Eine Grammatik-Testumgebung. In *Proceedings of KONVENS-94*, pages 427–430, Wien, 1994.

[KB82] Ronald Kaplan and Joan Bresnan. Lexical-functional grammar. A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge,MA, 1982.

[Lan87] *Language Craft Manual.* Pittsburgh, PA, 1987.

[Lie94] Christian Lieske. Object- and database-oriented integration of the CELEX lexical data in a system for natural language grammar engineering. (Diplomarbeit). Universität Koblenz-Landau, 1994.

[MPPV91] G. Marotta, M.T. Pazienza, F. Pettinelli, and P. Velardi. Extensive acquisition of syntactic knowledge in NLP systems. *Literary and Linguistic Computing*, 6(2):79–88, 1991.

[NKDvZ92] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The grammar workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and R. op den Akker, editors, *Proceedings of Second Twente Workshop on Language Technology*, pages 103–115, Twente, NL, 1992.

[PS87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Notes*. University of Chicago Press, Stanford, 1987.

[PS94] Carl Pollard and Ivan Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.

[Sch88] K. Schifferer. TAGDevEnv. Eine Werkbank für TAGs. In István Bátori, U. Hahn, M. Pinkal, and W. Wahlster, editors, *Computerlinguistik und ihre theoretischen Grundlagen*. Springer Verlag, Berlin, 1988.

[Vol94] Martin Volk. *Einsatz einer Testsatzsammlung im Grammar Engineering*. PhD thesis, Universität Koblenz-Landau, Koblenz, 1994.

[Whi87] John S. White. The research environment in the Metal project. In Sergei Nirenberg, editor, *Machine Translation: Theoretical and Methodological Issues*, pages 225–246. Cambridge University Press, Cambridge, 1987.